Schema Evolution Analysis for Embedded Databases

Shengfeng Wu, Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521
{wus,neamtiu}@cs.ucr.edu

***Abstract*—Dynamic software updating research efforts have mostly been focused on updating application code and in-memory state. As more and more applications use embedded databases for storage, dynamic updating solutions will have to support changes to embedded database schemas. The first step towards supporting dynamic updates to embedded database schemas is understanding how these schemas change—so far, schema evolution studies have focused on large, enterprise-class databases. In this paper we propose an approach for automatically extracting embedded schemas from regular applications, e.g., written in C and C++, and automatically computing how schemas change as applications evolve. To showcase our approach, we perform a long-term schema evolution study on four popular open source programs that use embedded databases: Firefox, Monotone, BiblioteQ and Vienna. Our study spans 18 cumulative years of schema evolution and reveals that change patterns and frequency in embedded databases differ from schema changes in enterprise-class databases that formed the object of prior studies. Our platform can be used for performing long-term, large-scale embedded schema evolution studies that are potentially beneficial to dynamic updating and schema evolution researchers.**

I. Introduction

Database evolution, and software evolution in general, are facts of life: to remain competitive, software providers are under increasing pressure to frequently release software updates, to fix bugs and add new features. The most common update practice today is to stop the application, apply the update, and restart. Unfortunately, this stop/restart update method is at odds with providing a seamless user experience. Restarting a program, or rebooting the machine is disruptive for a mobile or desktop application and could be intolerable for a server with high-availability requirements.

Many mobile, desktop, and server applications have recently started shifting from storing data in custom file formats towards storing data using a database management system contained within the application, hidden from the user, requiring no maintenance; these systems are called *Embedded Databases* (ED) [1]. An ED enables an application to manage data in a safer and more flexible manner, while at the same time rendering the data easier to query. The shift towards EDs is evidenced by the popularity of SQLite, a server-less, zero-config SQL engine [2]. SQLite is extremely popular: it is part of all major mobile platforms, Google Android, Apple iOS, Symbian and BlackBerry; operating systems (e.g., Mac OS X, Solaris 10, OpenSolaris); user space applications (e.g., Apple Mail, Safari, iTunes, Firefox, McAfee antivirus [3]); web applications [4]. An estimate by the SQLite development team puts the number of SQLite installations upwards of 500 million [5].

A stop/restart update for applications with EDs involves stopping the application (which necessarily implies shutting down the database), applying the update, and restarting. Ideally, however, we want to be able to update the application code and the database dynamically (on-the-fly). Many solutions exist already for dynamically updating "standard" applications written in C, C++, and Java. For example, dynamic software updating (DSU) systems such as Ginseng [6], Upstare [7], or Jvolve [8] allow on-the-fly updates to code and in-memory data. However, because of their focus on updating code and in-memory state, dynamic updating systems are insufficient for performing online upgrades to applications that require database updates. For example, in the update from Firefox 3.0b2 to 3.0b3, the attribute `user_title` was deleted from table `moz_history`. If we use a DSU system for Firefox, we can update the code, but the information stored in the database remains at the old version, which will lead to schema incompatibility, and possibly an update failure.

The broad goal of our work is to close this gap by permitting safe, dynamic schema updates to EDs; the first step towards this goal is understanding how ED schemas evolve. To that end, in this paper we present a system we constructed, called SCVD (which stands for Schema extraCtion and eVolution analysis for embedded Databases), that helps us understand and quantify schema evolution in EDs.

SCVD is a tool that automates schema extraction and schema evolution analysis for EDs. Given the evolution time frame for an application (set of releases), SCVD automatically retrieves the source code for all these releases, extracts the ED schemas from each version of the application code, compares schemas for successive versions, and presents the schema evolution results in an easy-to-understand manner.

Researchers and developers are equally likely to benefit from using SCVD. Via large-scale evolution studies, researchers can understand how applications with EDs evolve, and construct effective frameworks for supporting safe dynamic schema updates. Using SCVD, developers can compare old and new applications to find out when and how to correctly

978-1-4244-9196-4/11/$26.00 © 2011 IEEE

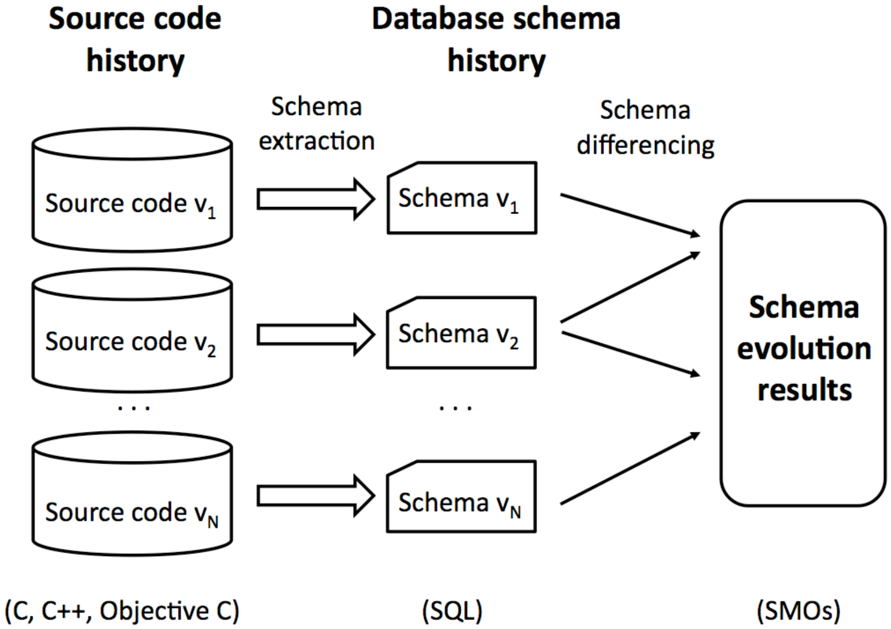

Fig. 1. High-level overview of SCVD.

migrate an ED from the old schema to the new schema. In fact, our own prior work [9] shows that ED schema updates are sometimes ad-hoc, i.e., when an application changes the ED schema, the new application version does not check the schema version in the ED prior to executing queries. Without migration after updating the application, the new queries will run against the old schema, which can lead to data loss or runtime errors. Note that the compiler does not detect such errors, as schemas and queries are simply strings interspersed throughout the application code.

For large, enterprise-class databases, researchers and practitioners have proposed many solutions to reconcile applications and schema updates, such as query rewriting [10], schema versioning and temporal querying [11], [12], schema mapping [13], [14], schema matching [15], or editioning views [16]. However, these approaches might not be suitable for ED schema evolution for several reasons; for example, they require SMOs (*Schema Modification Operators* [17], [10]) to be specified by the developers; also, schema matching/mapping/versioning implementations might impose significant overhead which is problematic for mobile and resource-constrained systems.

Our paper is structured as follows. In Section II we present the architecture and operation of SCVD. In Section III we present the results of a long-term schema evolution study—18 cumulative years—we conducted using SCVD on four popular open source programs that use EDs: Firefox, Monotone, BiblioteQ and Vienna. We found that ED schemas change less frequently than enterprise-class DB schemas, and that in EDs, deletions are more frequent. In Section IV we discuss possible threats to the validity of our study. Finally, in Section V we present future research directions.

In short, this paper makes the following contributions:

- An approach for extracting ED schemas and detecting schema evolution.
- A study of ED schema evolution for four popular applications over more than 18 cumulative years.

II. APPROACH

A high-level overview of SCVD is provided in Figure 1. SCVD starts with the release history of an application, extracts the database schemas embedded in the application, compares the schemas, and produces a tally of schema evolution results.

The *source code history* extractor takes as input a list of versions or tags, and checks out/downloads a corresponding list of source code versions we want to analyze. For example, the input for our Firefox study was a list of 308 CVS tags that correspond to 308 revisions of Firefox; the input for Monotone was a list of 48 Monotone official versions available on the Web as `.tar.gz`'s.

The *schema extractor* is a module that extracts the database schemas embedded in source code; it currently supports C, C++, and Objective C. The extractor uses language-specific patterns (expressed as regular expressions) to match schema structures in application code and extract a list of tables, along with their schemas, into .sql files, e.g., for application version X we create a corresponding schema_X.sql. The difficulty of extracting tables and their schemas from application code should not be understated. In the projects we analyzed, tables and schemas move from file to file as a result of new releases, some of their host C/C++ files disappear, some tables and schemas are embedded in .sql that is executed directly, etc. Therefore, correctly reconstituting the entire lineage (evolution history) of each embedded table was quite challenging.

TABLE I
EVOLUTION TIME FRAME AND SCHEMA CHANGE DETAILS (AS ABSOLUTE NUMBERS AND PERCENTS).

Application	Time frame	Table changes		Attribute changes					Changes per year
		CREATE TABLE	DROP TABLE	ADD COLUMN	DROP COLUMN	Type change	Init change	Key change	
Firefox	10/02/2004–11/21/2008	5 (4.2%)	26 (21.7%)	57 (47.5%)	28 (23.3%)	0 (0%)	3 (2.5%)	1 (0.7%)	29.3
Monotone	04/06/2003–06/13/2010	11 (20.4%)	17 (31.5%)	14 (25.9%)	10 (18.5%)	0 (0%)	0 (0%)	2 (3.7%)	7.6
BiblioteQ	03/15/2008–02/19/2010	4 (2.6%)	8 (5.2 %)	27 (17.5%)	28 (18.2%)	83 (53.9%)	0 (0%)	4 (2.6%)	80.2
Vienna	06/29/2005–09/03/2010	1 (7.1%)	0 (0%)	13 (92.9%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	2.7
Total		*21 (6.1%)*	*51 (14.9%)*	*111 (32.5%)*	*66 (19.3%)*	*83 (24.3%)*	*3 (0.9%)*	*7 (2%)*	

The *schema differencing* module computes differences between two .sql schema files and tallies the results. The module is based on `mysqldiff` [18], an open-source schema migration assistant. As `mysqldiff` is designed to work with the MySQL SQL dialect, we translate the database schemas from the SQLite SQL dialect into MySQL dialect before passing the schemas to `mysqldiff`.

SCVD also computes, for each embedded SQL table, the originating source file (C, C++, or Objective C), as well as the version interval for that table, i.e., the version the table was introduced in and the version the table was deleted in; however, we omit presenting these results as the focus of this paper is on schema changes.

III. SCHEMA EVOLUTION STUDY

A. Applications Examined

Previous works by Mandalapa [19], and Curino et al. [20] have studied schema evolution for TikiWiki/Joomla/Slash/MediaWiki (23 years, cumulative), and Wikipedia (4.5 years), respectively. The domain of these studies is enterprise-class databases, rather than EDs. Therefore, to understand how EDs evolve and are used in practice, we used SCVD to perform a schema evolution study, covering a cumulative 18 years of evolution, on four popular open source programs: Firefox, Monotone, BiblioteQ and Vienna. Firefox [1] is the widely-used open source browser; it is written mainly in C and C++, and uses SQLite to store the browsing history, input forms, cookies, etc. Monotone [2] is a version control system; it is written in C++ and uses SQLite to store file revisions, deltas, and branch information. BiblioteQ [3] is a catalog and library management suite written in C++. Vienna [4] is a popular RSS/Atom newsreader for Mac OS X; it is written in Objective C and uses SQLite to store news folders and messages.

B. Results

Our study was focused on table- and attribute-level changes that affect update safety. In Table I we summarize the results. The analysis time frame for each application is provided in column 2. The rest of the columns summarize schema changes as SMOs and attribute-level changes.

`CREATE TABLE` and `DROP TABLE` represent table addition and deletion, while `ADD COLUMN` and `DROP COLUMN` represent column addition and deletion. There is no SMO that refers to changes to attribute types, e.g., from `INTEGER` to `VARCHAR`, so we report those as “Type change”. Similarly, there is no SMO for changes to attribute initializers so we report those as “Init change”. Finally, we report changes to the key status of an attribute as “Key change”; these changes correspond to Shneiderman and Thomas’s [17] `PROMOTE TO KEY` and `DEMOTE FROM KEY`.

For each application, we count the changes over the time span we studied, and provide both the total count and percentages. For example, in Firefox, there were 5 `CREATE TABLE`s, 26 `DROP TABLE`s, 57 `ADD COLUMN`s, etc., which constitute 4.2%, 21.7%, 47.5%, etc. of the Firefox changes. The last row sums up the changes across all applications; as we can see, when considering all applications, the most frequent operations were `ADD COLUMN` (32.5%), type changes (24.3%), `DROP COLUMN` (19.3%), `DROP TABLE` (14.9%), and `CREATE TABLE` (6.1%).

C. Discussion

Designing an effective on-the-fly schema update system for EDs crucially depends on understanding how EDs schemas change in practice. For example, a system that supports `DEMOTE FROM KEY` but does not support `DROP COLUMN` is not going to be very effective in practice, because key changes are rare. Therefore, we structure the discussion of our findings by comparing our results with results from prior work, along the following two dimensions:

1) Nature of changes: what schema changes are more frequent in EDs compared to enterprise-class databases?
2) Frequency and timing of changes: when, and how frequently, do ED schemas change?

Nature of changes: Based on the total number of changes (last row in Table I), we can infer that any schema update system must support `ADD COLUMN`, `DROP COLUMN`, `DROP TABLE`, and `CREATE TABLE`, as these changes are frequent in all applications. While type changes are frequent in BiblioteQ,[5] the other applications do not use them; therefore,

[1] http://mozilla.org

[2] http://monotone.ca/

[3] http://biblioteq.sourceforge.net/

[4] http://vienna-rss.org

[5] Most of them consisted of changing `LONGTEXT` to `TEXT` and `INTEGER` to `BIGINT`.

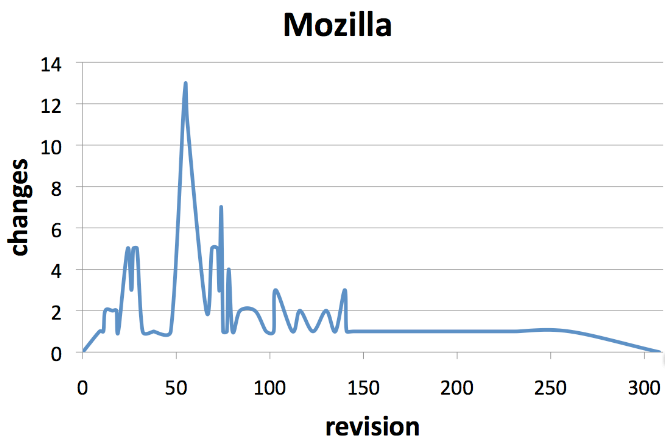

Fig. 2. Mozilla: frequency of schema changes over time.

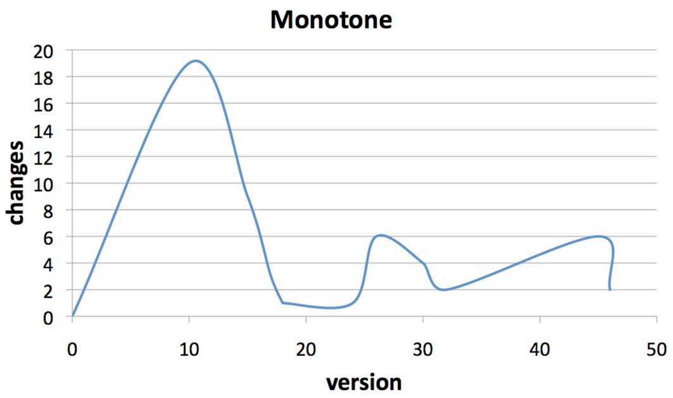

Fig. 3. Monotone: frequency of schema changes over time.

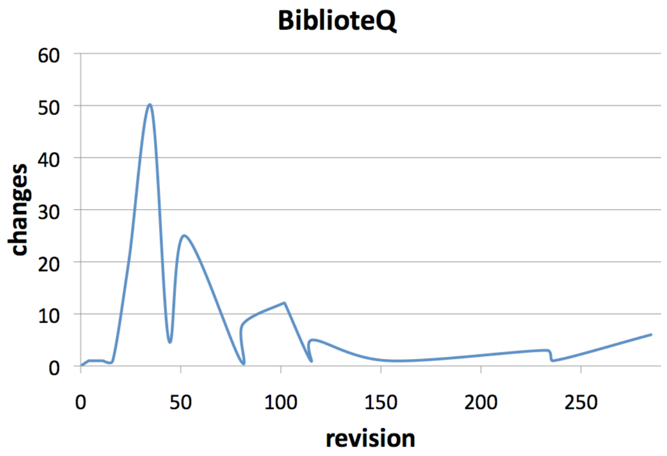

Fig. 4. BiblioteQ: frequency of schema changes over time.

we also computed totals ignoring the type changes in BiblioteQ, which yields the following distribution: `ADD COLUMN` (42.9%), `DROP COLUMN` (25.5%), `DROP TABLE` (19.7%), and `CREATE TABLE` (8.1%).

Our `CREATE TABLE` results are similar to those of Mandalapa [19] and Curino et al. [20] (their ratios are 8.7% and 8.9%, respectively). The same goes for `ADD COLUMN` (their ratios are 30.7% and 38.7%). However, we found many more `DROP TABLE`'s (their percentages are 8.7 and 3.3). Our `DROP COLUMN` percentages are similar to Curino et al. (26.4%), but much higher than Mandalapa's (6%). Mandalapa found that `CHANGE COLUMN` accounts for 25.6%, similar to us if we include the type changes in BiblioteQ, though note that Firefox, Monotone and Vienna have zero type changes and overall there is a low percentage of changes to initializers and keys.

The high numbers of column and table deletions we found lead us to believe that the ED systems we analyzed tend to undergo more restructuring, rather than exhibit continuous growth [20] as in the enterprise-class databases analyzed by prior work. The low number of column changes (excepting BiblioteQ) lead us to believe that supporting table and column additions/deletions is more important than supporting changes to column types, initializers and key status.

Frequency and timing of changes: In Figures 2, 3, and 4 we plot the timing and frequency of schema changes. In each graph, the x-axis represents program version/revision, while the y-axis represents the number of schema changes used in that version (i.e., changes compared to the prior version). As we can see, the trends are similar across programs: schemas tend to change more in the beginning, and the database structure stabilizes over time because later versions have fewer changes. This suggests that on-the-fly schema updates are necessary, especially in the beginning of a program's lifetime. Interestingly, Curino et al. [20] have found that for Wikipedia, the number of SMOs does not decrease over time.

In terms of frequency (changes per year), our applications' change rate varies between 2.7 and 80.2 per year, as can be seen in the last column of Table I. This suggests that ED schema change rate is generally lower than the schema change rate for enterprise-class DBs: prior work has found the annual change frequency to be 58.7 for Wikipedia [20], 56 for Joomla, 95 for Slash, 91.7 for TikiWiki, and 57.7 for MediaWiki [19].

IV. Threats to Validity

We now discuss possible threats to the validity of our study. One possible source of errors is missing tables, e.g., by using inadequate patterns that fail to extract all the tables from the C/C++/Objective C code. Another possible source of errors is renamings: we use a coarse-grained model that counts table and column renamings as a deletion followed by an addition.

External validity, i.e., the results generalize to other systems, is also threatened in our study. We have chosen a fairly broad range of open-source applications to examine: a browser, a version control server, an RSS reader, and a library storage and query system. Other application categories, or proprietary applications, or applications written in languages other than C/C++/Objective C, might display different schema evolution characteristics.

V. Future Work

We plan to extend this work in several directions. Straightforward extensions involve supporting other "host" languages besides C, Objective C, and C++ (for example Python, PHP and Ruby) and studying a broader range of applications that contain EDs.

A more challenging extension is detecting more complicated SMOs such as DISTRIBUTE TABLE, PARTITION TABLE, MERGE TABLE, RENAME TABLE, and RENAME COLUMN. We plan to use query contexts to help with detecting these SMOs. For example if a query:

```
SELECT A,B,C FROM foo
```

in the old application version is replaced with the query:

```
SELECT foo.A, foo.B, bar.C  FROM foo,bar WHERE foo.A = bar.A
```

in the new application version, this suggests a DISTRIBUTE TABLE was used to split the table foo with schema (A,B,C) into tables foo and bar with schemas (A,B) and (A,C), respectively.

VI. Related Work

Curino et al. [20] have studied schema evolution for Wikipedia from April 2003 to November 2007. Their study provides both micro- and macro-classification of changes. The micro-classifications correspond to SMO syntax, which is a superset of the changes we investigate; in particular, we do not collect data on DISTRIBUTE TABLE, MERGE TABLE, COPY COLUMN, and MOVE COLUMN. Macro-classifications include changes to indexes, keys, types, syntax and engine; we decided to only consider changes to keys, types and initializers, as these have a direct impact on schema upgrade safety. They also compute the query success rate when running a query against preceding and following schema versions; we do not compute this, though we envision doing so in the future when we start analyzing queries for context information.

Mandalapa [19] developed a tool called SEATS which can analyze schema evolution in large, enterprise-class DBs. They performed a large-scale schema evolution study on popular content management systems (CMS): TikiWiki (7 years), Joomla (2 years), Slash (8 years) and MediaWiki (6 years). In addition to table and column changes, they also compute the query failure rate, i.e., percent of new queries that would fail when run on the old schema. Our approach targets a different class of applications (EDs with schemas embedded in C/C++/Objective C, as opposed to CMSs with external databases and explicit schemas) and we have investigated 18 years (cumulative) of evolution, as opposed to their 23 cumulative years.

Sjøberg [21] presents a schema evolution study on a health management system over 1.5 years; their findings are similar to ours, i.e., most frequent changes are column additions/deletions and table additions/deletions.

Our own prior work [9] presents a schema evolution study on Firefox and Monotone over smaller time frames than the time frames considered in this paper; BiblioteQ and Vienna were not part of that study. The main difference between the two works reside in schema extraction and computing schema changes: in the prior work, extraction and change computation was manual. In this paper, extraction and differencing are automatic, which is key to scalability. In fact, when we compared the results from this paper with our prior work, we identified several schema changes that the prior, manual approach has been missing.

VII. Conclusions

In this paper we propose an approach for automating schema extraction and schema change detection in embedded databases, a previously-unexplored application domain. Using our toolset, we performed a schema evolution study on four popular open source applications that employ embedded databases, and showed that embedded databases tend to change differently than the enterprise-class databases that have been the object of prior studies. Our approach can be used for performing long-term, large-scale schema evolution studies that are potentially beneficial to dynamic updating and schema evolution researchers alike.

Acknowledgments

We thank Pamela Bhattacharya and the anonymous referees for their helpful comments on this paper.

References

[1] Wikipedia, "Embedded database," http://en.wikipedia.org/wiki/ Embedded_database.

[2] D. R. Hipp, "Sqlite," http://www.sqlite.org/.

[3] SQLite Team, "Well-known users of SQLite," http://www.sqlite.org/famous.html.

[4] P. Bhattacharya and I. Neamtiu, "Dynamic updates for web and cloud applications," in *APLWACA*, 2010, pp. 21–25.

[5] SQLite team, "Most widely deployed SQL database," http://www.sqlite.org/mostdeployed.html.

[6] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *PLDI*. New York, NY, USA: ACM Press, 2006, pp. 72–83.

[7] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 31–31.

[8] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: a vm-centric approach," in *PLDI*. New York, NY, USA: ACM, 2009, pp. 1–12.

[9] D.-Y. Lin and I. Neamtiu, "Collateral evolution of applications and databases," in *ERCIM Workshop on Software Evolution/International Workshop on Principles of Software Evolution*, August 2009, pp. 31–40.

[10] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: the prism workbench," *VLDB*, vol. 1, no. 1, pp. 761–772, 2008.

[11] H. J. Moon, C. A. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo, "Managing and querying transaction-time databases under schema evolution," *VLDB*, vol. 1, no. 1, pp. 882–895, 2008.

[12] C. Plattner, A. Wapf, and G. Alonso, "Searching in time," in *SIGMOD*. New York, NY, USA: ACM, 2006, pp. 754–756.

[13] C. Yu and L. Popa, "Semantic adaptation of schema mappings when schemas evolve," in *VLDB*. VLDB Endowment, 2005, pp. 1006–1017.

[14] Y. Velegrakis, R. J. Miller, and L. Popa, "Mapping adaptation under evolving schemas," in *VLDB*. VLDB Endowment, 2003, pp. 584–595.

[15] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[16] Oracle, *Edition-Based Redefinition*, http://www.oracle.com/technology/ deploy/availability/pdf/edition_based_redefinition.pdf.

[17] B. Shneiderman and G. Thomas, "An architecture for automatic relational database system conversion," *ACM Trans. Database Syst.*, vol. 7, no. 2, pp. 235–257, 1982.

[18] A. Spiers, “mysqldiff,” http://adamspiers.org/computing/mysqldiff/.

[19] V. Mandalapa, “A framework for understanding schema evolution in web information systems,” Master’s thesis, Arizona State University, 2009.

[20] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, “Schema evolution in wikipedia - toward a web information system benchmark.” in *ICEIS (1)*, 2008.

[21] D. Sjøberg, “Quantifying schema evolution,” in *Information and Software Technology*, vol. 35, no. 1, January 1993, pp. 35–44.